Our Ref.: 723-974

U.S. PATENT APPLICATION

Inventor(s): Farhad FOULADI
Winnie Wing Yan YEUNG
Howard CHENG

Invention: GRAPHICS PROCESSING SYSTEM WITH ENHANCED MEMORY CONTROLLER

NIXON & VANDERHYE P.C.
ATTORNEYS AT LAW
1100 NORTH GLEBE ROAD
8TH FLOOR
ARLINGTON, VIRGINIA 22201-4714
(703) 816-4000
Facsimile (703) 816-4100

SPECIFICATION

GRAPHICS PROCESSING SYSTEM WITH ENHANCED MEMORY CONTROLLER

Specification

This application is filed in accordance with 35 U.S.C. §119(e)(1) and claims the benefit of the provisional application Serial No. 60/226,894 filed on August 23, 2000, entitled "Graphics Processing System With Enhanced Memory Controller."

Field of the Invention

The present invention relates to computer graphics, and more particularly to interactive graphics systems such as home video game platforms. Still more particularly this invention relates to a memory controller for use in such an interactive graphics system that controls resource access to main memory.

Related Applications

This application is particularly related to application serial number 60/226,886, entitled "Method and Apparatus For Accessing Shared Resources" (atty. dkt no. 723-754), which is hereby incorporated by reference. This application is also related to the following applications identified below (by title and attorney docket number), which focus on various aspects of the graphics processing described herein. Each of the following applications are hereby incorporated herein by reference.

- provisional Application No. 60/161,915, filed October 28, 1999 and its corresponding utility Application No. 09/465,754, filed December 17, 1999, both entitled "Vertex Cache For 3D Computer Graphics",

- provisional Application No. 60/226,912, filed August 23, 2000 and its corresponding utility Application No. _____, filed _____ (atty. dkt. no. 723-959), both entitled "Method and Apparatus for Buffering Graphics Data in a Graphics System ",
- provisional Application No. 60/226,889, filed August 23, 2000 and its corresponding utility Application No. _____, filed _____ (atty. dkt. no. 723-958), both entitled "Graphics Pipeline Token Synchronization",
- provisional Application No. 60/226,891, filed August 23, 2000 and its corresponding utility Application No. _____ , filed _____ (atty. dkt. no. 723-961), both entitled "Method And Apparatus For Direct and Indirect Texture Processing In A Graphics System",
- provisional Application No. 60/226,888, filed August 23, 2000 and its corresponding utility Application No. _____, filed _____ (atty. dkt. no. 723-968), both entitled "Recirculating Shade Tree Blender For A Graphics System",
- provisional Application No. 60/226,892, filed August 23, 2000 and its corresponding utility Application No. _____, filed _____ (atty. dkt. no. 723-960), both entitled "Method And Apparatus For Efficient Generation Of

Texture Coordinate Displacements For Implementing Emboss-Style Bump Mapping In A Graphics Rendering System",

- provisional Application No. 60/226,893, filed August 23, 2000 and its corresponding utility Application No. _____ filed _____ (atty. dkt. no. 723-962), both entitled "Method And Apparatus For Environment-Mapped Bump-Mapping In A Graphics System",
- provisional Application No. 60/227,007, filed August 23, 2000 and its corresponding utility Application No. _____, filed _____ (atty. dkt. no. 723-967), both entitled "Achromatic Lighting in a Graphics System and Method",
- provisional Application No. 60/226,900, filed August 23, 2000 and its corresponding utility Application No. _____, filed _____ (atty. dkt. no. 723-964), both entitled "Method And Apparatus For Anti-Aliasing In A Graphics System",
- provisional Application No. 60/226,910, filed August 23, 2000 and its corresponding utility Application No. _____, filed _____ (atty. dkt. no. 723-957), both entitled "Graphics System With Embedded Frame Buffer Having Reconfigurable Pixel Formats",

- utility Application No. 09/585,329, filed June 2, 2000, entitled "Variable Bit Field Color Encoding" (atty. dkt. no. 723-749),
- provisional Application No. 60/226,890, filed August 23, 2000 and its corresponding utility Application No. _____, filed _____ (atty. dkt. no. 723-956), both entitled "Method And Apparatus For Dynamically Reconfiguring The Order Of Hidden Surface Processing Based On Rendering Mode",
- provisional Application No. 60/226,915, filed August 23, 2000 and its corresponding utility Application No. _____ filed _____ (atty. dkt. no. 723-973), both entitled "Method And Apparatus For Providing Non-Photorealistic Cartoon Outlining Within A Graphics System",
- provisional Application No. 60/227,032, filed August 23, 2000 and its corresponding utility Application No. _____, filed _____, (atty. dkt. no. 723-954), both entitled "Method And Apparatus For Providing Improved Fog Effects In A Graphics System",
- provisional Application No. 60/226,885, filed August 23, 2000 and its corresponding utility Application No. _____, filed _____, (atty. dkt. no. 723-969), both entitled "Controller Interface For A Graphics System",

- provisional Application No. 60/227,033, filed August 23, 2000 and its corresponding utility Application No. _____, filed _____ (atty. dkt. no. 723-955), both entitled "Method And Apparatus For Texture Tiling In A Graphics System",
- provisional Application No. 60/226,899, filed August 23, 2000 and its corresponding utility Application No. _____, filed _____ (atty. dkt. no. 723-971), both entitled "Method And Apparatus For Pre-Caching Data In Audio Memory",
- provisional Application No. 60/226,913, filed August 23, 2000 and its corresponding utility Application No. _____, filed _____ (atty. dkt. no. 723-965), both entitled "Z-Texturing",
- provisional Application No. 60/227,031, filed August 23, 2000 entitled "Application Program Interface for a Graphics System" (atty. dkt. no. 723-880),
- provisional Application No. 60/227,030, filed August 23, 2000 and its corresponding utility Application No. _____ , filed _____ (atty. dkt. no. 723-963), both entitled "Graphics System With Copy Out Conversions Between Embedded Frame Buffer And Main Memory",
- provisional Application No. 60/226,886, filed August 23, 2000 and its corresponding utility Application No. _____, filed _____ (atty. dkt. no.

723-970), both entitled "Method and Apparatus for Accessing Shared Resources",

- provisional Application No. 60/226,884, filed August 23, 2000 and its corresponding utility Application No. _____, filed _____ (atty. dkt. no. 723-972), both entitled "External Interfaces For A 3D Graphics and Audio Coprocessor",
- provisional Application No. 60/226,914, filed August 23, 2000 and its corresponding utility Application No. _____, filed _____, (atty. dkt. no. 723-966), both entitled " Low Cost Graphics System With Stitching Hardware Support For Skeletal Animation", and
- provisional Application No. 60/227,006, filed August 23, 2000 and its corresponding utility Application No. _____, filed _____ (atty. dkt. no. 723-953), both entitled " Shadow Mapping In A Low Cost Graphics System".

Background And Summary Of The Invention

Many of us have seen films containing remarkably realistic dinosaurs, aliens, animated toys and other fanciful creatures. Such animations are made possible by computer graphics. Using such techniques, a computer graphics artist can specify how each object should look and how it should change in appearance over time, and a computer then models the objects and displays them on a display such as your television or a computer screen. The computer takes care of performing the many tasks required to make sure that each part of the displayed

image is colored and shaped just right based on the position and orientation of each object in a scene, the direction in which light seems to strike each object, the surface texture of each object, and other factors.

Because computer graphics generation is complex, computer-generated three-dimensional graphics just a few years ago were mostly limited to expensive specialized flight simulators, high-end graphics workstations and supercomputers. The public saw some of the images generated by these computer systems in movies and expensive television advertisements, but most of us couldn't actually interact with the computers doing the graphics generation. All this has changed with the availability of relatively inexpensive 3D graphics platforms such as, for example, the Nintendo 64® and various 3D graphics cards now available for personal computers. It is now possible to interact with exciting 3D animations and simulations on relatively inexpensive computer graphics systems in your home or office.

In generating exciting 3D animations and simulations on relatively inexpensive computer graphics systems, it is important to efficiently control access to main memory among competing resources. Any such access control system is burdened with considerable constraints. For example, the main application program executing CPU, which is but one of many resources seeking access to main memory, must be granted memory access with a fixed memory read latency allowing for high speed execution of instructions. Accordingly, such a CPU should be awarded high priority access to main memory. In order to generate exciting graphics, certain graphics related resources seeking memory access must likewise be guaranteed high speed access to memory sufficient for the graphics processing to be rapidly completed.

The present invention is embodied in the disclosed illustrative memory controller described herein, which performs a wide range of memory control related functions including arbitrating between various competing resources seeking access to main memory. Other tasks performed by the unique memory controller include handling memory latency and bandwidth requirements of the resources requesting memory access, buffering writes to reduce turn around, refreshing main memory, protecting main memory using programmable registers, and numerous other functions.

In controlling memory access between resources seeking to read from and write to main memory, the memory controller minimizes switching between memory reads and memory writes to avoid wasting memory bandwidth due to idle cycles resulting from such switching and thereby enhancing memory access time. The illustrative memory controller minimizes such switching by incorporating a unique write buffering methodology that uses a "global" write queue which queues write requests from various diverse competing resources to reduce read/write switching. In this fashion, multiple competing resources for memory writes are combined into one resource from which write requests are obtained.

The memory controller in accordance with the illustrative embodiment described herein, advantageously optimizes access to main memory taking into account resource memory latency and bandwidth requirements.

The memory controller described herein uniquely resolves memory coherency issues to avoid accessing stale data from memory due to reading data from a main memory address location prior to when that same location had been updated by a write operation. Coherency issues are addressed both within a single resource that has both read and write capability and difference resources. The

exemplary embodiment addresses such coherency issues by efficiently flushing buffers associated with a resource. For example, a resource that is writing to main memory may send a flush signal to the memory controller to indicate that the resource's write buffer should be flushed. In accordance with an exemplary implementation, the memory controller generates a flush acknowledge handshake signal to indicate to competing resources that data written to main memory is actually stored in main memory rather than in an associated resource buffer.

Brief Description Of The Drawings

These and other features and advantages provided by the invention will be better and more completely understood by referring to the following detailed description of presently preferred embodiments in conjunction with the drawings, of which:

Figure 1 is an overall view of an example interactive computer graphics system;

Figure 2 is a block diagram of the Figure 1 example computer graphics system;

Figure 3 is a block diagram of the example graphics and audio processor shown in Figure 2;

Figure 4 is a block diagram of the example 3D graphics processor shown in Figure 3;

Figure 5 is an example logical flow diagram of the Figure 4 graphics and audio processor;

Figures 6A and 6B are block diagrams depicting memory controller and competing resources coupled thereto;

Figure 7 is an exemplary block diagram depicting various resources accessing main memory.

Figure 8 is a more detailed block diagram of the memory controller shown in Figures 6A and 6B;

Figure 9 illustrates a memory controller address path;

Figure 10 illustrates a memory controller read data path;

Figure 11 is a block diagram showing an exemplary set of communication signals exchanged between the memory controller and the processor interface (PI);

Figure 12 is a block diagram showing an exemplary set of communication signals exchanged between the memory controller and video interface;

Figure 13 is a block diagram showing an exemplary set of communication signals exchanged between the memory controller and cache/command processor;

Figure 14 is a block diagram showing an exemplary set of communication signals exchanged between the memory controller and the texture unit 500;

Figure 15 is a block diagram showing an exemplary set of communication signals exchanged between the memory controller and the pixel engine (PE) 700; and

Figures 16A and 16B show example alternative compatible implementations.

Detailed Description Of Example Embodiments Of The Invention

Figure 1 shows an example interactive 3D computer graphics system 50. System 50 can be used to play interactive 3D video games with interesting stereo sound. It can also be used for a variety of other applications.

In this example, system 50 is capable of processing, interactively in real time, a digital representation or model of a three-dimensional world. System 50 can display some or all of the world from any arbitrary viewpoint. For example, system 50 can interactively change the viewpoint in response to real time inputs from handheld controllers 52a, 52b or other input devices. This allows the game player to see the world through the eyes of someone within or outside of the world. System 50 can be used for applications that do not require real time 3D interactive display (e.g., 2D display generation and/or non-interactive display), but the capability of displaying quality 3D images very quickly can be used to create very realistic and exciting game play or other graphical interactions.

To play a video game or other application using system 50, the user first connects a main unit 54 to his or her color television set 56 or other display device by connecting a cable 58 between the two. Main unit 54 produces both video signals and audio signals for controlling color television set 56. The video signals are what controls the images displayed on the television screen 59, and the audio signals are played back as sound through television stereo loudspeakers 61L, 61R.

The user also needs to connect main unit 54 to a power source. This power source may be a conventional AC adapter (not shown) that plugs into a standard home electrical wall socket and converts the house current into a lower DC voltage signal suitable for powering the main unit 54. Batteries could be used in other implementations.

The user may use hand controllers 52a, 52b to control main unit 54. Controls 60 can be used, for example, to specify the direction (up or down, left or right, closer or further away) that a character displayed on television 56 should move within a 3D world. Controls 60 also provide input for other applications (e.g., menu selection, pointer/cursor control, etc.). Controllers 52a and 52b can take a variety of forms. In this example, controllers 52 shown each include controls 60a or 60b such as joysticks, push buttons and/or directional switches. Controllers 52 may be connected to main unit 54 by cables or wirelessly via electromagnetic (e.g., radio or infrared) waves.

To play an application such as a game, the user selects an appropriate storage medium 62 storing the video game or other application he or she wants to play, and inserts that storage medium into a slot 64 in main unit 54. Storage medium 62 may, for example, be a specially encoded and/or encrypted optical and/or magnetic disk. The user may operate a power switch 66 to turn on main unit 54 and cause the main unit to begin running the video game or other application based on the software stored in the storage medium 62. The user may operate controllers 52a, 52b to provide inputs to main unit 54. For example, operating a control 60a, 60b may cause the game or other application to start. Moving other controls 60a, 60b can cause animated characters to move in different directions or change the user's point of view in a 3D world. Depending upon the particular software stored within the storage medium 62, the various controls 60a, 60b on a controller 52a, 52b can perform different functions at different times.

Example Electronics of Overall System

Figure 2 shows a block diagram of example components of system 50. The primary components include:

- a main processor (CPU) 110,
- a main memory 112, and
- a graphics and audio processor 114.

In this example, main processor 110 (e.g., an enhanced IBM Power PC 750) receives inputs from handheld controllers 52 (and/or other input devices) via graphics and audio processor 114. Main processor 110 interactively responds to user inputs, and executes a video game or other program supplied, for example, by external storage media 62 via a mass storage access device 106 such as an optical disk drive. As one example, in the context of video game play, main processor 110 can perform collision detection and animation processing in addition to a variety of interactive and control functions.

In this example, main processor 110 generates 3D graphics and audio commands and sends them to graphics and audio processor 114. The graphics and audio processor 114 processes these commands to generate dynamic visual images on display 59 and high quality stereo sound on stereo loudspeakers 61R, 61L or other suitable sound-generating devices.

Example system 50 includes a video encoder 120 that receives image signals from graphics and audio processor 114 and converts the image signals into analog and/or digital video signals suitable for display on a standard display device such as a computer monitor or home color television set 56. System 50 also includes an audio codec (compressor/decompressor) 122 that compresses and decompresses digitized audio signals and may also convert between digital and analog audio signaling formats as needed. Audio codec 122 can receive audio inputs via a buffer 124 and provide them to graphics and audio processor 114 for processing (e.g., mixing with other audio signals the processor generates and/or receives via a

streaming audio output of mass storage access device 106). Graphics and audio processor 114 in this example can store audio related information in an audio memory 126 that is available for audio tasks. Graphics and audio processor 114 provides the resulting audio output signals to audio codec 122 for decompression and conversion to analog signals (e.g., via buffer amplifiers 128L, 128R) so they can be reproduced by loudspeakers 61L, 61R.

Graphics and audio processor 114 has the ability to communicate with various additional devices that may be present within system 50. For example, a parallel digital bus 130 may be used to communicate with mass storage access device 106 and/or other components. A serial peripheral bus 132 may communicate with a variety of peripheral or other devices including, for example:

- a programmable read-only memory and/or real time clock 134,
- a modem 136 or other networking interface (which may in turn connect system 50 to a telecommunications network 138 such as the Internet or other digital network from/to which program instructions and/or data can be downloaded or uploaded), and
- flash memory 140.

A further external bus 142, which may, by way of example only, be a serial bus, and may be used to communicate with additional expansion memory 144 (e.g., a memory card) or other devices. Connectors may be used to connect various devices to busses 130, 132, 142.

Example Graphics And Audio Processor

Figure 3 is a block diagram of an example graphics and audio processor 114. Graphics and audio processor 114 in one example may be a single-chip ASIC

(application specific integrated circuit). In this example, graphics and audio processor 114 includes:

- a processor interface 150,
- a memory interface/controller 152,
- a 3D graphics processor 154,
- an audio digital signal processor (DSP) 156,
- an audio memory interface 158,
- an audio interface and mixer 160,
- a peripheral controller 162, and
- a display controller 164.

3D graphics processor 154 performs graphics processing tasks. Audio digital signal processor 156 performs audio processing tasks. Display controller 164 accesses image information from main memory 112 and provides it to video encoder 120 for display on display device 56. Audio interface and mixer 160 interfaces with audio codec 122, and can also mix audio from different sources (e.g., streaming audio from mass storage access device 106, the output of audio DSP 156, and external audio input received via audio codec 122). Processor interface 150 provides a data and control interface between main processor 110 and graphics and audio processor 114.

As will be explained in detail below, memory interface 152 provides a data and control interface between graphics and audio processor 114 and memory 112. In this example, main processor 110 accesses main memory 112 via processor interface 150 and memory interface 152 that are part of graphics and audio processor 114. Peripheral controller 162 provides a data and control interface between graphics and audio processor 114 and the various peripherals mentioned

above. Audio memory interface 158 provides an interface with audio memory 126.

Example Graphics Pipeline

Figure 4 shows a graphics processing system including a more detailed view of an exemplary Figure 3 3D graphics processor 154. 3D graphics processor 154 includes, among other things, a command processor 200 and a 3D graphics pipeline 180. Main processor 110 communicates streams of data (e.g., graphics command streams and display lists) to command processor 200. Main processor 110 has a two-level cache 115 to minimize memory latency, and also has a write-gathering buffer 111 for uncached data streams targeted for the graphics and audio processor 114. The write-gathering buffer 111 collects partial cache lines into full cache lines and sends the data out to the graphics and audio processor 114 one cache line at a time for maximum bus usage.

Command processor 200 receives display commands from main processor 110 and parses them -- obtaining any additional data necessary to process them from shared memory 112 via memory controller 152. The command processor 200 provides a stream of vertex commands to graphics pipeline 180 for 2D and/or 3D processing and rendering. Graphics pipeline 180 generates images based on these commands. The resulting image information may be transferred to main memory 112 for access by display controller/video interface unit 164 -- which displays the frame buffer output of pipeline 180 on display 56.

Figure 5 is a block logical flow diagram portraying illustrative processing performed using graphics processor 154. Main processor 110 may store graphics command streams 210, display lists 212 and vertex arrays 214 in main memory

112, and pass pointers to command processor 200 via processor/bus interface 150. The main processor 110 stores graphics commands in one or more graphics first-in-first-out (FIFO) buffers 210 it allocates in main memory 110. The command processor 200 fetches:

- command streams from main memory 112 via an on-chip FIFO memory buffer 216 that receives and buffers the graphics commands for synchronization/flow control and load balancing,
- display lists 212 from main memory 112 via an on-chip call FIFO memory buffer 218, and
- vertex attributes from the command stream and/or from vertex arrays 214 in main memory 112 via a vertex cache 220.

Command processor 200 performs command processing operations 200a that convert attribute types to floating point format, and pass the resulting complete vertex polygon data to graphics pipeline 180 for rendering/rasterization. A programmable memory arbitration circuitry 130 (see Figure 4) arbitrates access to shared main memory 112 between graphics pipeline 180, command processor 200 and display controller/video interface unit 164.

Figure 4 shows that graphics pipeline 180 may include:

- a transform unit 300,
- a setup/rasterizer 400,
- a texture unit 500,
- a texture environment unit 600, and
- a pixel engine 700.

Transform unit 300 performs a variety of 2D and 3D transform and other operations 300a (see Figure 5). Transform unit 300 may include one or more matrix memories 300b for storing matrices used in transformation processing 300a. Transform unit 300 transforms incoming geometry per vertex from object space to screen space; and transforms incoming texture coordinates and computes projective texture coordinates (300c). Transform unit 300 may also perform polygon clipping/culling 300d. Lighting processing 300e also performed by transform unit 300b provides per vertex lighting computations for up to eight independent lights in one example embodiment. Transform unit 300 can also perform texture coordinate generation (300c) for embossed type bump mapping effects, as well as polygon clipping/culling operations (300d).

Setup/rasterizer 400 includes a setup unit which receives vertex data from transform unit 300 and sends triangle setup information to one or more rasterizer units (400b) performing edge rasterization, texture coordinate rasterization and color rasterization.

Texture unit 500 (which may include an on-chip texture memory (TMEM) 502) performs various tasks related to texturing including for example:

- retrieving textures 504 from main memory 112,
- texture processing (500a) including, for example, multi-texture handling, post-cache texture decompression, texture filtering, embossing, shadows and lighting through the use of projective textures, and BLIT with alpha transparency and depth,
- bump map processing for computing texture coordinate displacements for bump mapping, pseudo texture and texture tiling effects (500b), and

- indirect texture processing (500c).

Texture unit 500 outputs filtered texture values to the texture environment unit 600 for texture environment processing (600a). Texture environment unit 600 blends polygon and texture color/alpha/depth, and can also perform texture fog processing (600b) to achieve inverse range based fog effects. Texture environment unit 600 can provide multiple stages to perform a variety of other interesting environment-related functions based for example on color/alpha modulation, embossing, detail texturing, texture swapping, clamping, and depth blending..

Pixel engine 700 performs depth (z) compare (700a) and pixel blending (700b). In this example, pixel engine 700 stores data into an embedded (on-chip) frame buffer memory 702. Graphics pipeline 180 may include one or more embedded DRAM memories 702 to store frame buffer and/or texture information locally. Z compares 700a' can also be performed at an earlier stage in the graphics pipeline 180 depending on the rendering mode currently in effect (e.g., z compares can be performed earlier if alpha blending is not required). The pixel engine 700 includes a copy operation 700c that periodically writes on-chip frame buffer 702 to main memory 112 for access by display/video interface unit 164. This copy operation 700c can also be used to copy embedded frame buffer 702 contents to textures in the main memory 112 for dynamic texture synthesis effects. Anti-aliasing and other filtering can be performed during the copy-out operation. The frame buffer output of graphics pipeline 180 (which is ultimately stored in main memory 112) is read each frame by display/video interface unit 164. Display controller/video interface 164 provides digital RGB pixel values for display on display 56.

Figures 6A and 6B are illustrative block diagrams depicting memory controller 152 (Figures 3 and 4) and various resources coupled thereto which compete for access to main memory 112. Main memory 112 may, for example, comprise an SRAM, such as a 1TSRAM, manufactured by Mosys Corporation, which automatically performs internal refresh operations. Memory interface controller 152 provides a data and control interface between main processor 110, graphics and audio processor 114 and main memory 112. Although memory controller 152 and graphics memory request arbitration 130 are depicted as separate components in Figure 4, in the illustrative implementation described below, memory controller 152 also includes graphics memory request arbitration 130.

As shown in Figures 6A and 6B, memory controller 152 is coupled to various competing resources seeking to access main memory 12. Such competing resources include processor interface (PI) 150 (which is coupled to main processor 110), audio DSP (DSP) 156, input/output interface (IO) 802, video interface (VI) 164, cache/command processor (CP) 200, texture unit (TC) 500, and pixel engine (PE) 700. In this exemplary embodiment, of these resources, processor interface 150, audio DSP 156 and IO interface 802 are operable to both read information from and write information to main memory 112. IO interface 802 is operable to itself arbitrate and interface with a wide range of input/output devices such as modem, DVD interface and has relatively low memory bandwidth requirements. In the present illustrative embodiment, video interface 164, cache/command processor 200, and texture unit 500 are operable to only read information from main memory 112, and pixel engine 700 is operable to only write information to main memory 112.

Memory controller 152 performs various memory controller tasks including: 1) arbitrating among, for example, the 7 ports depicted in Figures 6A and 6B for access to main memory 112, 2) granting memory access taking into account memory latency and bandwidth requirements of the resources requesting memory access, 3) buffering writes to reduce access turn around, 4) refreshing main memory 112 when necessary, and 5) protecting main memory 112 using programmable registers. Although the illustrative embodiment shown in Figures 6A and 6B, depicts 7 ports seeking memory access, as will be appreciated by those skilled in the art, there may be greater or fewer than 7 ports in any given implementation. Moreover, the bus/signal line widths shown in Figure 6B and other Figures (as well as other implementation details) are presented for illustrative purposes only and should in no way be construed as limiting the scope of the present invention. Memory controller 152 performs arbitration among the identified ports and sends requests to the main memory 112. In the illustrative embodiment, memory controller 152 and all of its inputs and outputs run at 200 MHz. A 128 bit 200 MHz data path is up clocked at up to 400MHz through the 4-channel Memory Access Control (MAC) block to permit communication with a 400MHz external 1TSRAM memory. The MAC stores data received over respective 32 bit paths and clocks out the data at the appropriate clock rate. The address and control signals shown in Figure 6B are directly connected to the IO pins. The particular signaling used to communicate with main memory 112 is not a part of this invention.

In accordance with one exemplary embodiment of the memory controller resource arbitration methodology (and as further described in co-pending application Serial No 60/226,886, entitled "Method and Apparatus For Accessing

Shared Resources", atty. dkt no. 723-754, which application is hereby incorporated herein by reference), a bandwidth control is uniquely associated with each of the above-identified resources to permit an application programmer to control the bandwidth allocation of, for example, the 3.2 gigabyte's main memory 112 bandwidth. For example, programmable bandwidth control registers are respectively associated with command processor 200 and texture unit 500, which may be utilized to allocate more of the available main memory bandwidth to the command processor 200 than to texture unit 500. In this fashion, sophisticated users are able to tune the above-identified competing interface drivers to their particular application needs to get better overall performance. Accordingly, for each of the above-identified competing interfaces, a register is utilized to control its allocation of memory bandwidth to ensure that for every n number of clock cycles, a request for memory arbitration will be granted. Thus, for each interface, a filter is utilized which will, for example, slow down a request for main memory access if a particular interface is generating a large number of requests at a time when other interfaces are likewise generating requests. Alternatively, if main memory 112 is idle, and no other unit is contending for memory access, then such a request for access may be granted. The filter may define the speed at which requests for a given interface may be granted when other requests from different interfaces are being simultaneously entertained.

Memory controller 152 controls a wide range of graphics data related requests for main memory 112 involving for example:

- 3D graphics processor 154 (specifically, command processor 200, texture unit 500 and pixel engine 700),
- main processor 110 via processor interface 150,

- audio DSP 156,
- display controller 164, and
- peripheral controller 162 for various I/O units (e.g., mass storage access device 106)

Figure 7 illustrates some of the typical operations involved in these "requestors" competing for access to main memory. The arrows in Figure 7 represent the following operations:

1. Loading texture images from mass storage device 62 (e.g., DVD) to main memory 112 for a new image, game sector or level, or other application sequence

2. Loading geometry vertex arrays from mass storage device 62 to main memory for a new image, game sector or level, or other application sequence

3. Dynamic rendering of texture maps by main processor 110 or graphics processor 154

4. Dynamic generation or modification of vertex arrays by main processor 110

5. Main processor 110 animating lights and transformation matrices for consumption by graphics processor 154

6. Main processor 110 generating display lists for consumption by graphics processor 154

7. Main processor 110 generating graphics command streams

8. 3D graphics processor 154 reading graphics command stream

9. 3D graphics processor 154 reading display lists

10. 3D graphics processor 154 accessing vertices for rendering

11. 3D graphics processor 154 accessing textures for rendering

In the illustrative implementation, the graphics processor 114 has several data memory requirements including alignment requirements for the following types of data: texture and texture lookup table images, display lists, graphics FIFO and the external frame buffer. These data objects should be aligned because the graphics processor 114 is very fast; data from the main memory 112 is transferred in 32-byte chunks. Data alignment allows for simple and fast hardware.

On other data objects, such as vertex, matrix and light arrays, in an exemplary embodiment additional hardware support eliminates the need for coarse alignment (these are 4-byte aligned). There are a large number of these data objects, and the memory consumption of each object is potentially low, so relaxing alignment restrictions helps to conserve memory.

In accordance with the illustrative implementation, multiple processors and hardware blocks can update main memory. In addition, the CPU 110 and graphics processor 114 contain various data caches. Since the hardware does not maintain coherency of the data in main memory and various associated caches, there are various potential sources of coherency problems including when the CPU modifies or generates data destined for the graphics processor 114, when the CPU writes data through its write-gather buffer to cached memory, and when loading new data destined for the graphics processor 114 from the DVD into main memory. Coherency problems may occur if the main memory used to store the data in these two latter cases were used for other graphics data.

When the DVD loads data, the DVD API automatically invalidates the loaded main memory portion that resides in the CPU data cache. This feature

provides a safe method for programmers to modify the DVD loaded data without worrying about CPU data cache coherency. This DVD API feature activates by default; it can be deactivated by the programmer.

The graphical data loaded by DVD may contain textures and vertices that have been already formatted for the graphics processor 114 to render. Therefore, invalidation of the vertex cache and texture cache regions may be necessary.

The CPU 110 has two means of writing to main memory: the write-gather buffer and the CPU cache hierarchy. The write-gather buffer is normally used to "blast" graphics commands into memory without affecting the cache. As a result, information sent through the write-gather buffer is not cache coherent. Care must be taken when using the write-gather buffer to avoid writing to areas of memory that maybe found in the CPU cache. The cache flushing instructions shown below maybe used to force data areas out of the CPU cache.

If the CPU generates or modifies graphics data through its cache, the following memory types may end up containing stale data:

- Main memory.
- graphics processor 114 vertex cache and texture cache regions.

To send the correct data to the graphics processor 114, in accordance with the exemplary embodiment, there is a need to flush the CPU data cache as well as invalidate the graphics processor 114 vertex or texture cache. The CPU typically animates data one frame ahead of the graphics processor 114, so efficient techniques to maintain data coherency include:

- Grouping all the CPU-modified graphics data in main memory sequentially, so that the block data cache flush is efficient.

- Invalidating the vertex cache, as well as the entire texture cache, at the beginning of each graphics frame.

These operations are mentioned by way of illustrating some of the many operations involving reading and writing to main memory 112. Among other things, memory controller 152 arbitrates among the ports involved in main memory reading and writing operations.

Figure 8 is a more detailed block diagram of memory controller 152. As shown in Figure 8, memory controller 152 includes individual "local" interfaces associated with each of the competing resources shown in Figures 6A and 6B. A controller pi interface 150I interfaces with processor interface 150, controller DSP interface 156I interfaces with audio DSP 156, controller io interface 802I interfaces with input output interface 802, controller video interface 164I interfaces with video interface 164, controller cp interface 200I interfaces with command processor 200, controller tc interface 500I interfaces with texture unit 500, and interface pe 700I interfaces with pixel engine 700. Memory controller 152 is coupled to main memory via external memory control 829, which is shown in further detail in Figure 16 described below. External memory control as shown in Figure 16 generates a read/write control signal which switches the bidirectional memory bus between read and write states.

Focusing, for illustration purposes on the texture coordinate interface 500I, this interface is coupled to the read only texture unit 500 shown in Figures 6A and 6B. TC interface 500I (like each of the local interfaces coupled to resources which read from main memory) includes a read queue (RQ2 shown in Figure 9) for queuing read requests and associated memory addresses received from its associated resource, texture unit 500. Memory controller interfaces pe, dsp, io, and

pi also respectively include a local write queue WQ0-4 as shown in Figure 9 for queuing write requests.

Turning back to Figure 8, arbitration control 825 includes the control logic for implementing the arbitration methodology, which is described in further detail below and in the above-identified co-pending application entitled "Method and Apparatus for Accessing Shared Resources" (Attorney Docket No. 723-754) which has been incorporated herein by reference. Arbitration control 825 is alerted to the presence of, for example, the receipt of a read request in texture interface 500I. Similarly, interfaces 200I, 700I, 150I, 156I, 164I, 802I and 829 are operatively coupled to arbitration control 825 for arbitration of competing memory access requests. As will be explained further below, arbitration control 825 upon receipt of read requests from, for example, memory TC interface 500I and DSP interface 156I (if, for example, 500I and 156I were the only competing resources) may award a first memory cycle to texture unit TC and the next memory cycle to DSP 156, etc. Thus, the read requests may be granted on a round robin basis. Arbitration controller 825 is aware of all pending requests and grants as described in the above-identified co-pending patent application and as set forth further below.

As suggested by the read data path illustrated in Figure 10, texture unit TC has a high bandwidth requirement (e.g., see the 128 bit GFX data path which is the same width as the main memory data path). The texture unit thus may be granted a request without wasting memory bandwidth. DSP, as shown in Figure 10, has a 64 bit bandwidth and will be awarded priority by the arbitration control 825 in a manner designed not to waste memory cycles.

The arbitration control 825 may, for example, be a state machine which sequences through states that implement the arbitration methodology described below. As explained in detail in the above-identified copending patent application, the arbitration control 825 is controlled in part by bandwidth dial registers such that when (for example) there is a request for memory access from texture unit 500, the request may be effectively suppressed. Thus, in a video game in which there is a large amount of texture data, the system may be tuned to adjust the bandwidth to optimize it for that particular game's memory access needs.

More specifically, as stated above, each of the read "masters" (i.e., a resource seeking to access main memory 112) is associated with a respective corresponding one of read queues RQ1 to RQ6 for queuing read addresses for reading from main memory 112. Each of the write masters seeking to access main memory 112 is associated with a respective corresponding one of write queues WQ1 to WQ4 for queuing write addresses and corresponding data for writing to main memory 112. Arbitration control 825 uses a predetermined arbitration process to allocate main memory access among the read queues RQ1 to RQ6 and to control which write requests among the write queues WQ1 to WQ4 are provided to global write buffer WQ0. The rate at which at least some of the requests are fed into this arbitration process is controllable in accordance with the settings of programmable bandwidth dial registers. By appropriately setting the dial registers for a particular operation, sophisticated users can tune the flow of requests to the arbitration process to improve system performance for that operation.

By collecting the write requests into the global write buffer WQ0, read to write and write to read switching may be reduced, thereby minimizing the dead memory cycles that result when the main memory is changed from one type of

operation to the other. While write requests are supplied to global write buffer WQ0, read requests are processed in accordance with the arbitration process. The main memory data path is generally switched from a read to a write state when the global write buffer queue WQ0 is filled to a certain level or if a main processor read request matches an entry in the global write buffer. This switchover results in a flushing of the global write buffer WQ0 to write data to specified addresses of main memory 112.

As mentioned above, the dial registers control the memory bandwidth for the corresponding master. For example, if an accumulator to which the contents of command processor dial register are added every memory cycle is less than 1.00, even if there is a pending command processor request, the arbitration scheme grants memory access to another master until enough cycles elapse so that the contents of the accumulator is equals to or greater than 1.00, or until there is no pending request from any other masters. Memory controller 152 preferably does not permit the main memory 112 to be in an idle state because of dial register settings. The dial registers affect the arbitration scheme by masking requests from masters until the accumulator corresponding to the dial register of that master equals 1.00.

Thus, bandwidth dial registers influence the memory usage by some of the major memory "hogs". The read dials control the frequency with which the masters participate in the arbitration process and access memory. The write dials are for control flow and can slow down the writing device by throttling the writes into global write buffer WQ0. As noted, arbitration preferably does not allow the memory to be idle if there are outstanding read requests that not being allowed due

to the settings of the bandwidth dials. In this case, a round robin scheme is used among the requestors that are being throttled.

In the example system, all reads are single cache-line (32 bytes). Thus, it takes two cycles of 200 MHz to read the cache line and a new read can be performed every 10 nanoseconds. Reads from main processor 110 have the highest priority, with round robin arbitration among the rest of the requestors. Memory ownership is changed every 10 nanoseconds among the read requestors and refresh, but the write queue is always written in its entirety. The write queue initiates a request when it is filled to or above a certain level or if a main processor read request matches an entry in the write-buffer.

As shown in Figure 8, bandwidth dial registers and other registers identified specifically below are embodied in the memory controller's programmable memory registers 823. These registers, which are identified in detail below, are programmable by main CPU 110 to control a wide range of memory controller functions. Among the registers included in memory controller 152 are memory access performance related registers. For example, performance counter registers identify how many requests are received from particular competing resources. The performance counters are utilized to keep track of wasted memory cycles so that a determination may be made as to how effectively memory bandwidth has been allocated based upon an analysis of the performance counter registers. The performance counters may be utilized to differentiate between cycles which are necessarily lost in switching between read and write operations and idle time. As previously mentioned, cycles are wasted upon switching from a read to write, e.g., two idle cycles may result from such switching. The performance counters may be utilized to determine how well a particular application program is utilizing memory

bandwidth by subtracting from performance statistics memory cycles which must necessarily be utilized for read/write switching and refresh operations. By monitoring such performance, application programmers are advantageously enabled to design more efficient programs that make better use of memory.

Turning back to Figure 9, as stated above, each of the read queues RQ1 to RQ6, is resident in an associated interface in Figure 8. Thus, read RQ1, as suggested by the signal line designation in Figure 9 is resident in CP interface 200I. Similarly, write queue WQ1 (which in the illustrative embodiment queues eight requests) is resident only in PE interface 700I and is referred to herein as a "local" write queue buffer. Similarly, WQ2 through WQ4 are resident in the DSP IO and PI interfaces respectively and are local write queue buffers. WQ0 shown in Figure 9 is the multiple resource or "global" write buffer and is resident in the Figure 8 component wrbuf 827. The inputs to write buffer 827 shown in Figure 8 correspond to the inputs to WQ0 shown in Figure 9.

If, for example, multiple write requests are received in write buffer 827 at the same time, in accordance with an exemplary embodiment of the present invention, memory write buffer 827 may arbitrate among such write requests. Additionally, a dial register may be utilized in association with the global write buffer embodied in write buffer 827. In this fashion, a write request from PE or PI, through the use of a dial register, may be designated as a lower priority request by an application programmer. The global write buffer 827 is operatively coupled to the arbitration control 825 for arbitration of write requests.

The Figure 8 read requests from the respective read queues are directly coupled to arbitrator control 825 for arbitrating between received read requests. A request bus (which identifies whether a read from or write to main memory 112 is

to take place at an associated address) is associated with each of the resources which are seeking access to main memory 112. Memory controller 152 queues up received memory access requests and sends the request result to the requesting resource.

In the case of write requests, flow control is accomplished in part using the local write buffers, e.g., WQ1 to WQ4, such that a signal is sent to the associated resource writing data to main memory 112 when the local write buffer is full (or nearly full) to inform the resource to stop sending data.

Memory controller 152 is advantageously designed to minimize read to write switching, since lost memory cycles result from such switching due to the need to place the bus in the proper read or write state. Memory controller 152 minimizes such read or write switching by gathering the required writes into a global write buffer WQ0 resident in wrbuf 827. While write requests are buffered, read requests are processed by arbitration control 825 from different resources. When the write buffer WQ0 begins to get full, it will arbitrate with the read requests in round robin fashion. Thereafter, multiple writes are processed at essentially the same time from global write buffer WQ0,which is filled from multiple resources, e.g., WQ 1- WQ4. When the global write buffer WQ0 reaches a state where it is, for example, 75-80% full, memory controller 115 switches to a write state to initiate the flushing of the write buffer WQ0 to main memory 112 resulting in writing to the identified address locations.

Memory controller 152 utilizes three levels of write arbitration. The first level of arbitration occurs whereby write buffer control logic arbitrates with sources seeking to read information from memory. Another level of write arbitration occurs when the write buffers are not full. A third level of arbitration

occurs when coherency processing is required, whereby write buffers are flushed to resolve the coherency issue.

With respect to processing read requests, in accordance with an illustrative embodiment, a round robin read is performed among resources based upon resource request arbitration processing in light of, for example, the dial register contents for each resource as explained above.

The following table lists illustrative sizes for each of the read and write queues shown in Figure 9 :

Queue	interface	depth	Width addr,data,mask	explanation
RQ1	CP (read)	16	21	Match the max latency of a single access.
RQ2	TC (read)	16	21	Match the max latency of a single access.
RQ3	VI (read)	1	21	Single outstanding read request
RQ4	DSP (read)	1	21	Single outstanding read request
RQ5	IO (read)	1	21	Single outstanding read request
RQ6	PI (read)	6	23	Multiple outstanding read request + skid 2 extra address bits to transfer critical oct-byte first.
WQ1	PE (write)	8	21+128	Max transfer from WQ1 to WQ0 + skid
WQ2	DSP (write)	4	21+128 + 4	Single outstanding write, no skid
WQ3	IO (write)	4	21+128	Single outstanding write, no skid
WQ4	PI (write)	8	21+128 + 4	Max transfer from WQ4 to WQ0 + skid
WQ0	Global Write buffer	16	24+128 + 4	Deep enough to amortize memory data path read/write mode switch turn around....

Figure 10 shows the read data path from main memory 112 to the identified resource via memory access controllers 804, 806, 808. Even though there are 6 read requestors, there are only 3 read data paths going back to the devices, the 128 bit GFX path, the 64 bit system path, and the 64 bit CPU path. The exemplary implementation does not use a unique data path for each device, since data is not transferred on all data paths at the same time. The exemplary implementation does not use a single 128-bit data path, since 64 bit devices, which take 4 cycles to

receive data, are utilized. In-order to reduce the latency for CPU accesses, the CPU port was given its own path, and therefore two 64-bit paths and a 128-bit path have been utilized. The paths are connected as follows:

- the GFX path, 128 bits @ 200 MHz is connected to CP 200 and TC 500. The bus bandwidth (BW) is equal to memory BW.
- the CPU path, 64 bits @ 200 MHz is connected to the pi only. The bandwidth of this path is ½ of the bandwidth of memory 112.
- the system bus, 64 bits @ 200 MHz is connected to IO, DSP and VI. All these devices are low BW and can only issue single outstanding transactions. The bus BW is ½ of memory BW.

The number and BW of these buses have direct impact on the memory arbitration. For example, GFX path can continuously request data from memory, whereas CPU can request data only every other cache-line cycle (100mhz). And the same is true for system bus.

Data read from main memory 112 is sent back to a requesting resource in order. Accordingly, if a first request is followed by a second and other multiple outstanding requests, after arbitration of these requests, the requests are fulfilled in the order requested. Thus, data requested by requesting resource number 1 is routed followed by the data requested by requesting resource number 2, etc. Reads are expected by the CPU to be processed in order. The present design eliminates the need for hardware or software to perform reordering operations.

The memory controller advantageous is designed to efficiently respond to access requests in order to take full advantage of the main memory 112 use of a static RAM (SRAM) type of memory. As explained above, the example embodiment has a 1TSTRAM that provides near static RAM type access in the

context of a high density DRAM. The use of near SRAM access permits, for example, writing data to main memory 112 in the order desired because writing to one location in the SRAM takes the same time as writing to any other location no matter where in SRAM the data is to be stored. In contrast, when using DRAM, writes to memory must be scheduled in accordance with the memory refresh schedule to maximize speed preference. The use of an SRAM permits efficiently fulfilling requests in order at the price of having to maintain data coherency.

With respect to maintaining coherency (processor coherency in the preferred illustrative embodiment, since other resources may rely on flushes to guarantee read/write coherency), if a resource writes to an associated write buffer for thereafter writing data to main memory 112, and almost immediately thereafter an attempt is made to read such data from main memory 112, a coherency problem results due to the potential of reading stale data from main memory 112 instead of the updated data sought. The memory controller 152 addresses the coherency issue by ensuring that, for every read request, a check is made of the address to be read to ensure that such address does not appear in the write buffer. If the address is in the write buffer, then the write buffer needs to be flushed, i.e., copied to main memory, before the read operation is performed.

Certain of the resources such as, for example, the command processor CP 200 is a unidirectional resource such that it only performs read operations from main memory 112 and does not write to main memory 112. In the exemplary implementation, pixel engine PE only writes to main memory 112. Coherency issues particularly need to be addressed with CPU 110, since CPU 110 both reads and writes from and to main memory 112. Thus, with regard to CPU reads, the address to be read is compared to write buffer addresses and, as explained above, if

the address is in the write buffer, the write buffer is flushed, and then the read operation is performed. For example, if writes are performed by a particular resource to locations 0, 1, and 2, which addresses are resident in a write buffer, and an attempt is made to read from location 0, since location 0 is in the write buffer, the system should flush the write buffer contents before reading from location 0. Accordingly, in order to ensure against coherency errors within a device, such errors will only occur if the resource has both read and write capability.

However, it is also desirable for the memory controller 152, to ensure against coherency errors among different resources. Thus, if pixel engine 700 receives a command to copy information to main memory 112, the local write buffer associated with pixel engine 700 will contain both the data to be copied and an address location at which to write to main memory 112. If, for example, the video interface 164 as the texture unit 500 thereafter seeks to read data from the same address to which the pixel engine 700 is writing data, the illustrative memory controller 152 synchronizes these operations. Thus, in accordance with an exemplary embodiment of the present invention, any device/resource that is writing to main memory 112 sends a flush signal to memory controller 152 which indicates to memory controller 152 to empty the respective resource's write buffer. Memory controller 152 generates a signal indicating that such operation has been completed to thereby inform CPU 110 to enable, for example, display unit 164 to read data from such a memory location. The indication from memory controller 152 that data written to main memory 112 is actually stored in main memory 112 and not in a buffer gives any competing resource the opportunity to access such data. In accordance with this exemplary embodiment, coherency among devices is

guaranteed by the device writing to memory by virtue of the receipt from memory controller 152 of a flush acknowledge handshake signal.

In accordance with an exemplary embodiment of the present invention, since writes are delayed, there are various types of coherency protocols which are performed, several of which have been briefly described above. Such coherency protocols, which are described and summarized below include:

- Coherency between writes and reads from the same unit.
- Coherency between writes and reads for CPU.
- Coherency between writes by CPU and reads by CP in CP FIFO.
- Coherency between writes and reads from two different units.

RW coherency from the same unit

In the exemplary implementation, DSP, IO and PI can perform writes and reads. There is no hardware RW coherency for DSP or IO in accordance with an exemplary implementation. If each device needs to read back the data it wrote to main memory 112, it needs to explicitly flush the write-buffer. This is done by issuing a flush write buffer command and waiting for an acknowledge signal. The PI read requests on the other hand are checked against the write-buffer addresses. If there is a match, the write-buffer is flushed. Then the read will proceed. The write-buffer includes the individual write-buffer for the unit and the global write-buffer.

RW coherency from CPU

In order to handle CPU 110 write and read coherency, bypassing logic and write buffer flushing mechanism is used. For a read request from CPU, the read address is sent immediately to main memory 112 and there's not enough time for

RW coherency checking until one cycle later. If there's a match, since the read request has already been sent, the read data from the memory is aborted, then it will flush the write buffer, bypass and merge the write data and read data back to the CPU 110 at the end of the write buffer flush.

In the case that a read is followed by a write for the same address location, these two requests are dispatched into the read queue and write queue separately and memory controller 152 can not distinguish the order of these two requests. Therefore, read data may end up with the new write data instead of the original one as expected. CPU 110 configuration should be set accordingly to guarantee not to issue the write before the read data comes back.

RW coherency between other units.

In the exemplary implementation, there are 4 units that can write to memory: DSP, IO, PE and PI. Any time a device writes to memory, it needs to flush its write buffers explicitly, before signaling another device for reading the data. Each of these 4 interfaces has a 2-wire flush/acknowledge protocol to accomplish this. DSP, IO or PE will issue a flush at the end of a DMA write, before interrupting the CPU 110. This will guarantee that CPU 110 will access the desired data, upon read. CPU 110 also needs to perform an explicit flush when it sets up a buffer in main memory 112 and wants to initiate another device for a read DMA. Before starting the DMA, CPU 110 needs to perform a "sync" instruction. This instruction causes a sync bus cycle, which causes the memory controller 152 to flush the write buffer. Upon completion of the flush, the sync instruction is completed and CPU can start up a read DMA operation.

RW coherency between CPU/CP for CP FIFO

The memory controller also handles RW coherency between CPU writes and CP read for a command related buffer CP FIFO which is associated with external memory. PI will indicate whether the write request is for CP FIFO and memory controller will send CP the write request when the write data for CP FIFO has actually been committed to the main memory.

Turning back to Figure 8, memory controller 152 includes a set of memory registers 823 through which the memory controller may be programmably controlled to perform a wide range of memory control and arbitration functions. In the exemplary implementation of the present invention, all registers are written through the processor interface 150. Although a table of a illustrative memory controller registers is set forth below, the following registers may be categorized into groups as follows.

Memory protection/Interrupt enable registers

Four set of registers can be used for memory read, write or read/write protection by setting the read enable and/or write enable bits in MEM_MARR_CONTROL register shown in the illustrative register table below. For example, if a read address is within the range of MEM_MARR0_START and MEM_MARR0_END with MARR0 read disabled, it will set the MARR0 interrupt bit in MEM_INT_STAT register and MEM_INT_ADDRL, MEM_INT_ADDRH will have the read address that caused the interrupt. It can also cause an interrupt to the CPU if MARR0 interrupt enable bit is set in MEM_INT_ENBL register. Note that memory controller 152 is not going to terminate the read/write transaction to main memory 112 that causes the interrupt.

There is also an address interrupt that is generated if the request address is outside the current memory configuration range and within 64 Mbytes address space. If the request address is beyond 64 Mbytes, PI should generate the address interrupt and not send the request to memory controller.

Bandwidth dial registers

In the exemplary embodiment, there are dial registers for CP, TC, PE , CPU read and CPU write masters. These dial registers are used to lower the memory bandwidth for the associated master. For example, if the CP dial register contents when added to an associated accumulator is less than one, even if there's a pending CP request, the illustrative arbitration methodology will grant memory access to another master until CP dial register when added to the accumulator equals one or there's no other pending request from any other master. In other words, memory controller 152 never lets memory 112 be in an idle state because of the dial register settings. These dial registers indirectly affect the arbitration scheme by masking the request from that master if the dial register plus accumulator contents does not equal to 1.00. For further details, refer to the methodology described below and in more detail in the copending application entitled "Method and Apparatus For Accessing Shared Resources" (atty. dkt no. 723-754), which has been incorporated herein by reference.

Performance Counter registers

There is a request counter per master except CPU, which has separate read and write request counters. These counters are mainly used for collecting statistics about memory usage and bandwidth for different masters. There are two additional counters: MEM_FI_REQCOUNT for counting number of idle cycles

due to read/write bus turnaround overhead and MEM_RF_REQCOUNT for counting number of refresh cycles. All these counters will be clamped to max values when reached.

Data for Turnaround registers

There are 3 registers used for setting number of idle cycles for the data path turnaround: one for RD to RD from a different memory bank, one for RD to WR switching and one for WR to RD switching.

Memory Refresh and threshold registers

When the number of clocks reached the refresh count in refresh counter, a refresh request will be generated. If the memory is idle, memory will be granted to refresh cycles. However, if memory is non-idle, it will be granted only if the total number of refresh requests reaches the threshold value set in the memory refresh threshold register.

For purposes of illustrating an exemplary memory controller 152 register set, the following table shows example memory controller 152 registers.

Register address	Register name	Field	description
0x0 (r, w)	MEM_MARR0_START	[15:0]	Starting address of memory address range register 0 memory address (25:10)
0x2 (r, w)	MEM_MARR0_END	[15:0]	Ending address of memory address range register 0 memory address (25:10)
0x4 (r, w)	MEM_MARR1_START	[15:0]	Starting address of memory address range register 1 memory address (25:10)
0x6 (r, w)	MEM_MARR1_END	[15:0]	Ending address of memory address range register 1 memory address (25:10)
0x8 (r, w)	MEM_MARR2_START	[15:0]	Starting address of memory address range register 2 memory address (25:10)
0xa (r, w)	MEM_MARR2_END	[15:0]	Ending address of memory address range register 2 memory address (25:10)
0xc (r, w)	MEM_MARR3_START	[15:0]	Starting address of memory address range register 3

			memory address (25:10)
0xe (r, w)	MEM_MARR3_END	[15:0]	Ending address of memory address range register 3 memory address (25:10)
0x10 (r, w)	MEM_MARR_CONTROL	[7:0]	Control register for the MAR registers 3 to 0 0: MARR0 read enable (OK to read between MARR0_START and MARR0_END) ie MARR0_START <= Address < MARR0_END 1: MARR0 write enable(OK to write between MARR0_START and MARR0_END) ie MARR0_START <= Address < MARR0_END 2: MARR1 read enable 3: MARR1 write enable 4: MARR2 read enable 5: MARR2 write enable 6: MARR3 read enable 7: MARR3 write enable Default value: 0xff (okay to write or read)
0x12 (w)	MEM_CP_BW_DIAL	[8:0]	Format is 1.8. Every cycle this number is added to an accumulator that is initialized to 0. When bit 8, is set, then CP request is enabled and CP is allowed in arbitration. When set to 0x1.00, CP request is always enabled Default value: 0x1.00
0x14 (w)	MEM_TC_BW_DIAL	[8:0]	Format is 1.8. Every cycle this number is added to an accumulator that is initialized to 0. When bit 8, is set, then TC request is enabled and TC is allowed in arbitration. When set to 0x1.00, TC request is always enabled Default value: 0x1.00
0x16 (w)	MEM_PE_BW_DIAL	[8:0]	Format is 1.8. Every cycle this number is added to an accumulator that is initialized to 0. When bit 8, is set, then PE write request is enabled and PE write can be transferred from the first queue to the WQ0. When set to 0x1.00, PE write is always enabled Default value: 0x1.00
0x18 (w)	MEM_CPUR_BW_DIAL	[8:0]	Format is 1.8. Every cycle this number is added to an accumulator that is initialized to 0. When bit 8, is set, then CPU request is enabled and CPU read is allowed in arbitration. When set to 0x1.00, CPU read request is always enabled Default value: 0x1.00
0x1a (w)	MEM_CPUW_BW_DIAL	[8:0]	Format is 1.8. Every cycle this number is added to an accumulator that is initialized to 0. When bit 8, is set, then CPU write data is written into the write-buffer. When set to 0x1.00, CPU write data is accepted immediately Default value: 0x1.00
0x1c (r, w)	MEM_INT_ENBL	[4:0]	Interrupt enable register for MARRs and address out of range error 0: MARR0 interrupt enable 0 for disabled, 1: enabled 1: MARR1 interrupt enable 2: MARR2 interrupt enable 3: MARR3 interrupt enable 4: Address Error interrupt enable Default value: 0x00 (disable)
0x1e (r)	MEM_INT_STAT	[4:0]	Interrupt status register 0: MARR0 interrupt

			1: MARR1 interrupt 2: MARR2 interrupt 3: MARR3 interrupt 4: Address Error interrupt Reset value: 0x00
0x20 (w)	MEM_INT_CLR		Interrupt clear. Writing to register clears all interrupts.
0x22 (r)	MEM_INT_ADDRL	[15:0]	Bits 15:0 of the memory address that caused the interrupt.
0x24 (r)	MEM_INT_ADDRH	[9:0]	Bits 25:16 of the memory address that caused the interrupt.
0x26 (r, w)	MEM_REFRESH	[7:0]	Number of cycles between memory refresh Default value: 0x80 (128 cycles) If it is zero, it is a special case for not generating any refresh cycles. This must be used with mem_refresh_thhdA to have a minimum value of 1.
0x2c (r, w)	MEM_RDTORD	[0]	For back to back read in the memory development system: 0: One idle cycle asserted when switching between the two. 1: Two idle cycles asserted when switching between the two. Default value: 0
0x2e (r, w)	MEM_RDTOWR	[0]	For a read followed by a write: 0: Two idle cycles asserted for turn around. 1: Three idle cycles asserted for turn around. Default value: 0
0x30 (r, w)	MEM_WRTORD	[0]	For a write followed by a read: 0: No idle cycle asserted. 1: One idle cycle asserted. Default value: 0
0x32(r, w)	MEM_CP_REQCOUNTH	[15:0]	Upper 16 bits of the 32 bits counter for CP memory requests processed (31:16). Write 0 to clear counter.
0x34(r, w)	MEM_CP_REQCOUNTL	[15:0]	Lower 16 bits of the 32 bits counter for CP memory requests processed (15:0). Write 0 to clear counter.
0x36(r, w)	MEM_TC_REQCOUNTH	[15:0]	Upper 16 bits of the 32 bits counter for TC memory requests processed (31:16). Write 0 to clear counter.
0x38(r, w)	MEM_TC_REQCOUNTL	[15:0]	Lower 16 bits of the 32 bits counter for TC memory requests processed (15:0). Write 0 to clear counter.
0x3a(r, w)	MEM_CPUR_REQCOUNTH	[15:0]	Upper 16 bits of the 32 bits counter for CPU read requests processed (31:16). Write 0 to clear counter
0x3c(r, w)	MEM_CPUR_REQCOUNTL	[15:0]	Lower 16 bits of the 32 bits counter for CPU read requests processed (15:0). Write 0 to clear counter
0x3e(r, w)	MEM_CPUW_REQCOUNTH	[15:0]	Upper 16 bits of the 32 bits counter for CPU write requests processed (31:16). Write 0 to clear counter.
0x40(r, w)	MEM_CPUW_REQCOUNTL	[15:0]	Lower 16 bits of the 32 bits counter for CPU write requests processed (15:0). Write 0 to clear counter.
0x42(r, w)	MEM_DSP_REQCOUNTH	[15:0]	Upper 16 bits of the 32 bits counter for DSP write/read requests processed (31:16). Write 0 to clear counter.
0x44(r, w)	MEM_DSP_REQCOUNTL	[15:0]	Lower 16 bits of the 32 bits counter for DSP write/read requests processed (15:0). Write 0 to clear counter.
0x46(r, w)	MEM_IO_REQCOUNTH	[15:0]	Upper 16 bits of the 32 bits counter for IO write/read requests processed (31:16). Write 0 to clear counter.
0x48(r, w)	MEM_IO_REQCOUNTL	[15:0]	Lower 16 bits of the 32 bits counter for IO write/read requests processed (15:0). Write 0 to clear counter.
0x4a(r, w)	MEM_VI_REQCOUNTH	[15:0]	Upper 16 bits of the 32 bits counter for VI memory requests processed (31:16). Write 0 to clear counter.
0x4c(r, w)	MEM_VI_REQCOUNTL	[15:0]	Lower 16 bits of the 32 bits counter for VI memory requests processed (15:0). Write 0 to clear counter.

0x4e(r ,w)	MEM_PE_REQCOUNTH	[15:0]	Upper 16 bits of the 32 bits counter for PE memory requests processed (31:16). Write 0 to clear counter.
0x50(r ,w)	MEM_PE_REQCOUNTL	[15:0]	Lower 16 bits of the 32 bits counter for PE memory requests processed (15:0). Write 0 to clear counter.
0x52(r, w)	MEM_RF_REQCOUNTH	[15:0]	Upper 16 bits of the 32 bits counter for memory refresh requests processed (31:16). Write 0 to clear counter.
0x54(r, w)	MEM_RF_REQCOUNTL	[15:0]	Lower 16 bits of the 32 bits counter for memory refresh requests processed (15:0). Write 0 to clear counter.
0x56(r, w)	MEM_FI_REQCOUNTH	[15:0]	Upper 16 bits of the 33 bits counter for memory forced idle requests processed (32:17). Write 0 to clear counter. Increment by one every idle cycle.
0x58(r, w)	MEM_FI_REQCOUNTL	[15:0]	Lower 16 bits of the 33 bits counter for memory forced idle requests processed (16:1). Write 0 to clear counter. Increment by one every idle cycle.
0x5a (r, w)	MEM_DRV_STRENGTH	[10:0]	Drive Strength
0x5c (r, w)	MEM_REFRSH_THHD	[2:0]	Threshold for generating the refresh request when the total number of outstanding refresh requests exists. Default value: 0x2 In order to generate zero refresh cycles, this register must be set to be non-zero together with mem_refresh set to 0x0.

Turning back to the Figure 8 memory controller block diagram, as set forth above, memory controller 152 includes arbitration control 825 which operates to arbitrate memory access requests between the competing resources identified above. For further details regarding the arbitration control, reference should be made to copending application Serial No. 60/226,886, entitled "Method and Apparatus For Accessing Shared Resources", which has been incorporated herein by reference. All reads are single cache-line (32 bytes). It takes 2 cycles of 200Mhz to read the cache-line. Thus a new read can be performed every 10 nsec. CPU reads will have the highest priority, with round robin arbitration among the rest of the requestors. Memory ownership is changed every 10 nsec among the read requestors and refresh, but the write queue is always written in its entirety. Write queue initiates a request when it gets above a certain level or if a CPU read request address matches an entry in the write-buffer. In accordance with the illustrative embodiment, there are the following restrictions as to the frequency of requests:

- Two CPU reads can not occur back to back

- Two System reads can not occur back to back.
- During a 10-nsec refresh cycle, 2 rows are refreshed. One every 5 nsec.

- **BW dials**

As described above, BW dials are provided via the BW registers referenced above to influence the memory usage by some of the major memory users. There are dials for the following devices:

- CPU read
- CP read
- TC read
- CPU write
- PE write

The read dials control the frequency of the units to participate in arbitration and access memory. The write dials are for control flow and can slow down the writing device by throttling the writes into the main write buffer.

The arbitration methodology will not allow the memory 112 to be idle if there are outstanding read requests that are not being allowed due to the BW dial. In this case a round robin scheme is used among the requestors that are being throttled.

Read Queues Arbitration

CPU read has the highest priority except the following conditions:

CPU was the master for the previous access

CPU read dial knob does not equal to 1.00 and there are other requests by other masters with dial knob equals 1.00

Write Buffer is completely full and it is in the middle of the write cycles

Previous CPU read address matches a valid CPU write address in the PI local write buffer or global write buffer which will cause a write buffer flush

CP (or TC) read has the same priority as any other system masters (DSP, IO and VI) and hence arbitrates the memory in the round robin fashion with the system masters except the following conditions:

DSP, IO or VI was the master for the previous access which then cannot arbitrate again,

CP (or TC) read dial knob does not equal to 1.00 and there are other requests by other masters with dial knob equals to 1.00, it will then have a lowest priority

DSP (or IO or VI) read has the same priority as any other GFX masters (CP and TC) and hence arbitrates the memory in the round robin fashion with the GFX masters except the following conditions:

DSP (or IO or VI) was the master for the previous access which then cannot arbitrate again.

Write Buffer has lower priority then CPU, GFX or system masters except the following conditions:

Write Buffer is completely full and it will arbitrate with others in the round-robin fashion

CPU read address matches a write address in write buffer and it will have the highest priority

Any other masters with higher priority have the dial knob less than 1.00

Refresh has the lowest priority except the following conditions:

Number of total refresh requests reaches the threshold value, its priority will be bumped up to just below CPU read.

Any other masters with higher priority have the dial knob less than 1.00

Write Queues Arbitration

CPU, PE, DSP and IO are the four masters in the write queue. CPU writes has the highest priority and the other three masters arbitrate in the round-robin fashion except the following condition:

CPU write dial knob does not equal to 1.00 and there are other write masters with dial knob equals to 1.00

All these together will form the write buffer queue arbitrating the memory bandwidth with the read masters.

Each of the interfaces depicted in Figure 6A with memory controller 152 will now be described in further detail. Figure 11 is a block diagram showing an exemplary set of communication signals exchanged between memory controller 152 and processor interface (PI) 150. The interface shown in Figure 11 allows reads and writes to main memory 112 from CPU 110.

This interface supports multiple outstanding read requests. In the illustrative embodiment, a new read request can be issued every cycle and a new write request can be issued every 4 clocks (4 cycles to transfer the cache-line on the bus). The memory controller 152 performs flow control by asserting mem_pi_reqfull. Write data are not acknowledged. Read data are acknowledged with the transfer of the first oct-byte of the cache. If the request address is not 32B aligned, critical double

word will be returned first. All read data are processed in-order. Write data are buffered and delayed to increase memory efficiency. pi_mem_flush is asserted for one cycle to flush the write buffer. mem_pi_flush_ack is issued for one cycle to signal that the write buffer is flushed.

All interface control signals should be registered to any avoid timing problem due to long wire. For example, memory controller 152 should register the pi_mem_req signal first, and the generated mem_pi_ack signal should also be registered on both the memory controller 152 side and the Module 150 side.

However, due to the memory bandwidth and CPU performance reasons, pi_mem_addr will not be registered and will be sent immediately to the main memory, this will reduce one cycle of latency.

The signals exchanged in the illustrative embodiment between the memory controller 152 and the processor interface 150 are shown in the table below.

signal	**description**
pi_mem_addr[25:1]	Address of cache-line for read/write. Read is always double word aligned (critical double word first). Write is always 32 byte aligned. For main memory read, pi_mem_addr[25:3] will be used. For main memory write, pi_mem_addr[25:5] will be used. For memory register read/write, pi_mem_addr[8:1] will be used.
pi_*mem_req*	Asserted for one cycle to issue a cache-line read/write request. pi_mem_addr, pi_mem_fifoWr and pi_mem_rd are valid for that cycle. For a write request, the first Oct-byte of the data is also valid on the pi_mem_data bus in this cycle.
pi_ *mem_rd*	0 is write; 1 is read
pi_mem_reg	0: memory access ; 1: register access During register writes the lower 8 bits of the address holds the register address and pi_mem_data[63:48] hold the register value.
pi_mem_fifoWr	1: Memory writes for CP FIFO, valid only during pi_mem_req cycle.
mem_pi_reqfull	When this signal is asserted to 1, two more read and writes requests can be issued.
mem_ pi_ack	Asserted for one cycle to signal return of data from memory during read. Bytes 0 to 7 of the cache-line are sent in that cycle. Bytes 8-15, 16-23 and 24-31 are sent in the following cycles on the mem_pi_data bus. If the read address is not 32B aligned, critical double word will be returned first. No acknowledge signal will be asserted for memory writes. All ***read requests are processed in-order.***
mem_ pi _data[63:0]	8 byte bus to transfer data from memory. A cache-line is transferred on this bus in 4 back-to-back clocks. Critical double word will come first.
pi_*mem_data[63:0]*	8 byte bus to transfer data to memory. A cache-line is transferred on this bus in 4 back-to-back clocks. The pi_mem_msk[1:0] bits determine validity of the two 32-bit words.

pi_*mem_msk[1:0]*	32-bit word write mask bits for pi_mem_data[63:0]. pi_mem_msk[1] is write mask for pi_mem_data[63:32]. pi_mem_msk[0] is write mask for pi_mem_data[31:0]. Mask equals 0 for write enable.
pi_mem_flush	Asserted by the PI for one cycle to flush the write buffer in memory controller..
mem_pi_flush_ack	Asserted by mem for one cycle, when the write buffer is flushed.
mem_pi_int	Interrupt from mem to pi.
pi_mem_memrstb	Pi_mem_memrstb caused by power-on-reset or software-reset. Disabled by software writing to memrstb register in PI.

Turning next to the audio DSP 156/memory controller interface 152, the following table illustrates exemplary signals exchanged between these two components together with a signal description.

Error! Not a valid filename.

Turning next to the input-output interface 802/memory controller interface 152, the following table illustrates exemplary signals exchanged.

Signal name	description
io_memAddr[25:5]	Address of cache-line for read/write. Bits (4:0) are 0 and are not transmitted.
io_memReq	Asserted for one cycle to issue a cache-line read/write request. io_memAddr is valid for that cycle.
io_memRd	0 is write; 1 is read
mem_ioAck	Asserted for one cycle to signal return of data from memory during read. Bytes 7 to 0 of the cache-line are sent in that cycle. Bytes 15-8 , 23-16 and 31-24 are sent in the following cycles on the mem_ioData bus.
mem_ioData[63:0]	8 byte bus to transfer data from memory. A cache-line is transferred on this bus in 4 back-to-back clocks.
io_memData[63:0]	8 byte bus to transfer data to memory. A cache-line is transferred on this bus in 4 back-to-back clocks.
io_memFlushWrBuf	At the end of a write burst. This signal is asserted for one cycle, and causes the memory controller to flush the write buffer.
mem_ioFlushWrAck	This signal is asserted for one cycle when the memory controller has completed flushing the write buffer in response to the assertion of **io_memFlushWrBuf .**

With respect to the I/O interface 802/memory controller 152 signals, at most one outstanding transfer is permitted in the exemplary embodiment, i.e., the next transfer cannot start until the previous transfer completes (with mem_ioAck signal). There are at least two levels of write buffering on the memory controller side to buffer the write data. That is, the interface should be able to buffer the write

data from the Module 802 and delay issuing the acknowledge signal if the buffer is full.

Figure 12 is a block diagram showing an exemplary set of communication signals exchanged between memory controller 152 and video interface 164. The interface signals are described in the table below.

Error! Not a valid filename. Figure 13 is a block diagram showing an exemplary set of communication signals exchanged between memory controller 152 and cache/command processor 200. The table below illustrates exemplary signals exchanged. between these components.

Name:	Description
cp_mem_addr(25:5)	Address of cache-line for read. Bits (4:0) are 0 and are not transmitted.
cp_mem_req	Asserted for one cycle to issue a cache-line read request. cp_mem_addr is valid for that cycle.
mem_cp_reqFull	When asserted, the read request queue is almost full. Only 2 more requests can be sent.
mem_cp_ack	Asserted for one cycle to signal return of data from memory. Bytes 15 to 0 of the cache-line are sent in the next cycle. Bytes 31-16 are sent after two cycles.
mem_cp_fifoWr	Asserted for one cycle indicating a new data has been written to CP FIFO in the main memory by the CPU for CP to access.
mem_cp_data(127:0)	16 byte bus to transfer data from memory. A cache-line is transferred on this bus in 2 back-to-back clocks.

Figure 14 is a block diagram showing an exemplary set of communication signals exchanged between memory controller 152 and the texture unit 500. The table below illustrates exemplary signals exchanged between these components.

Error! Not a valid filename.

Figure 15 is a block diagram showing an exemplary set of communication signals exchanged between memory controller 152 and the pixel engine (PE) 700. It is used to transfer filtered frame buffer images to main memory for display. It also converts frame buffer format to texture format and writes it into main memory

112. The table below illustrates exemplary signals exchanged between these components.

Name:	Description
pe_mem_addr(25:5)	Address of the cache-line for write, bit 4 to bit 0 are always zero.
pe_mem_req	Asserted for one cycle to issue a cache-line write request. pe_mem_addr is valid for that cycle. The first ½ of the cache-line is on the data bus in this cycle.
pe_mem_data(127:0)	Data bus. The first ½ of the cache-line is transferred when pe_mem_req is asserted. The second ½ will be transferred in the next cycle. The 2 ½ cache-lines are always transferred in back to back cycles
pe_mem_flush	At the end of a write burst. This signal is asserted for one cycle, so that the memory controller will flush the write buffer.
mem_pe_flushAck	Memory controller will asserted this signal for one cycle after receiving pe_mem_flush and flushing the write buffer.
mem_pe_reqFull	When asserted, the write queue is almost full. If the signal is asserted in the same cycle as request, no more requests will come until the signal is de-asserted. When the signal is asserted in the cycle after request, one more request can be issued.

The memory controller 152 sends address and control signals directly to external memory. Among the control signals shown are the control signals for switching the bus between a read to a write state. The following table illustrates exemplary signals exchanged. between these components. Included among the signals are the read/write signals which are needed to switch the bidirectional memory bus from a read to write state.

Name	Direction	Bits	Description
Mema_topad	O	22	Memory address, bit 0 is always zero
Memrw_topad	O	1	0: Write 1: Read
Memadsb_topad	O	2	Bit 1 selects development memory, bit 0 selects main memory, active low
Memrfsh_topad	O	1	Refresh cycle
memdrvctl_topad	O	3	Drive strength control for address pads
memateb_topad	O	1	Active Terminator Enable, active low

Other Example Compatible Implementations

Certain of the above-described system components 50 could be implemented as other than the home video game console configuration described above. For example, one could run graphics application or other software written for system 50 on a platform with a different configuration that emulates system 50 or is otherwise compatible with it. If the other platform can successfully emulate, simulate and/or provide some or all of the hardware and software resources of system 50, then the other platform will be able to successfully execute the software.

As one example, an emulator may provide a hardware and/or software configuration (platform) that is different from the hardware and/or software configuration (platform) of system 50. The emulator system might include software and/or hardware components that emulate or simulate some or all of hardware and/or software components of the system for which the application software was written. For example, the emulator system could comprise a general purpose digital computer such as a personal computer, which executes a software emulator program that simulates the hardware and/or firmware of system 50.

Some general purpose digital computers (e.g., IBM or MacIntosh personal computers and compatibles) are now equipped with 3D graphics cards that provide 3D graphics pipelines compliant with DirectX or other standard 3D graphics command APIs. They may also be equipped with stereophonic sound cards that provide high quality stereophonic sound based on a standard set of sound commands. Such multimedia-hardware-equipped personal computers running emulator software may have sufficient performance to approximate the graphics and sound performance of system 50. Emulator software controls the hardware

resources on the personal computer platform to simulate the processing, 3D graphics, sound, peripheral and other capabilities of the home video game console platform for which the game programmer wrote the game software.

Figure 16A illustrates an example overall emulation process using a host platform 1201, an emulator component 1303, and a game software executable binary image provided on a storage medium 62. Host 1201 may be a general or special purpose digital computing device such as, for example, a personal computer, a video game console, or any other platform with sufficient computing power. Emulator 1303 may be software and/or hardware that runs on host platform 1201, and provides a real-time conversion of commands, data and other information from storage medium 62 into a form that can be processed by host 1201. For example, emulator 1303 fetches "source" binary-image program instructions intended for execution by system 50 from storage medium 62 and converts these program instructions to a target format that can be executed or otherwise processed by host 1201.

As one example, in the case where the software is written for execution on a platform using an IBM PowerPC or other specific processor and the host 1201 is a personal computer using a different (e.g., Intel) processor, emulator 1303 fetches one or a sequence of binary-image program instructions from storage medium 1305 and converts these program instructions to one or more equivalent Intel binary-image program instructions. The emulator 1303 also fetches and/or generates graphics commands and audio commands intended for processing by the graphics and audio processor 114, and converts these commands into a format or formats that can be processed by hardware and/or software graphics and audio processing resources available on host 1201. As one example, emulator 1303 may

convert these commands into commands that can be processed by specific graphics and/or or sound hardware of the host 1201 (e.g., using standard DirectX, OpenGL and/or sound APIs).

An emulator 1303 used to provide some or all of the features of the video game system described above may also be provided with a graphic user interface (GUI) that simplifies or automates the selection of various options and screen modes for games run using the emulator. In one example, such an emulator 1303 may further include enhanced functionality as compared with the host platform for which the software was originally intended.

Figure 16B illustrates an emulation host system 1201 suitable for use with emulator 1303. System 1201 includes a processing unit 1203 and a system memory 1205. A system bus 1207 couples various system components including system memory 1205 to processing unit 1203. System bus 1207 may be any of several types of bus structures including a memory bus or memory controller, a peripheral bus, and a local bus using any of a variety of bus architectures. System memory 1207 includes read only memory (ROM) 1252 and random access memory (RAM) 1254. A basic input/output system (BIOS) 1256, containing the basic routines that help to transfer information between elements within personal computer system 1201, such as during start-up, is stored in the ROM 1252. System 1201 further includes various drives and associated computer-readable media. A hard disk drive 1209 reads from and writes to a (typically fixed) magnetic hard disk 1211. An additional (possible optional) magnetic disk drive 1213 reads from and writes to a removable "floppy" or other magnetic disk 1215. An optical disk drive 1217 reads from and, in some configurations, writes to a removable optical disk 1219 such as a CD ROM or other optical media. Hard disk

drive 1209 and optical disk drive 1217 are connected to system bus 1207 by a hard disk drive interface 1221 and an optical drive interface 1225, respectively. The drives and their associated computer-readable media provide nonvolatile storage of computer-readable instructions, data structures, program modules, game programs and other data for personal computer system 1201. In other configurations, other types of computer-readable media that can store data that is accessible by a computer (e.g., magnetic cassettes, flash memory cards, digital video disks, Bernoulli cartridges, random access memories (RAMs), read only memories (ROMs) and the like) may also be used.

A number of program modules including emulator 1303 may be stored on the hard disk 1211, removable magnetic disk 1215, optical disk 1219 and/or the ROM 1252 and/or the RAM 1254 of system memory 1205. Such program modules may include an operating system providing graphics and sound APIs, one or more application programs, other program modules, program data and game data. A user may enter commands and information into personal computer system 1201 through input devices such as a keyboard 1227, pointing device 1229, microphones, joysticks, game controllers, satellite dishes, scanners, or the like. These and other input devices can be connected to processing unit 1203 through a serial port interface 1231 that is coupled to system bus 1207, but may be connected by other interfaces, such as a parallel port, game port Fire wire bus or a universal serial bus (USB). A monitor 1233 or other type of display device is also connected to system bus 1207 via an interface, such as a video adapter 1235.

System 1201 may also include a modem 1154 or other network interface means for establishing communications over a network 1152 such as the Internet. Modem 1154, which may be internal or external, is connected to system bus 123

via serial port interface 1231. A network interface 1156 may also be provided for allowing system 1201 to communicate with a remote computing device 1150 (e.g., another system 1201) via a local area network 1158 (or such communication may be via wide area network 1152 or other communications path such as dial-up or other communications means). System 1201 will typically include other peripheral output devices, such as printers and other standard peripheral devices.

In one example, video adapter 1235 may include a 3D graphics pipeline chip set providing fast 3D graphics rendering in response to 3D graphics commands issued based on a standard 3D graphics application programmer interface such as Microsoft's DirectX 7.0 or other version. A set of stereo loudspeakers 1237 is also connected to system bus 1207 via a sound generating interface such as a conventional "sound card" providing hardware and embedded software support for generating high quality stereophonic sound based on sound commands provided by bus 1207. These hardware capabilities allow system 1201 to provide sufficient graphics and sound speed performance to play software stored in storage medium 62.

While the invention has been described in connection with what is presently considered to be the most practical and preferred embodiment, it is to be understood that the invention is not to be limited to the disclosed embodiment, but on the contrary, is intended to cover various modifications and equivalent arrangements included within the scope of the appended claims.